MapReduce Performance in MongoDB Sharded Collections

Jaumin Ajdari, Brilant Kasami
Faculty of Contemporary Sciences and Technologies
South East European University (SEEU)
Tetovo, Macedonia

***Abstract*—In the modern era of computing and countless of online services that gather and serve huge data around the world, processing and analyzing Big Data has rapidly developed into an area of its own. In this paper, we focus on the MapReduce programming model and associated implementation for processing and analyzing large datasets in a NoSQL database such as MongoDB. Furthermore, we analyze the performance of MapReduce in sharded collections with huge dataset and we measure how the execution time scales when the number of shards increases. As a result, we try to explain when MapReduce is an appropriate processing technique in MongoDB and also to give some measures and alternatives to take when MapReduce is used.**

***Keywords*—NoSQL; big data; MapReduce; sharding; MongoDB**

I. Introduction

We live in the era of the Information Age. Everything is connected and online services are more and more oriented to user data gathering. Major companies process hundreds of petabytes daily at their servers and the computations have to be distributed across hundreds or thousands of machines in order to finish it in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures obscure the simple computations within a large amounts of complex code in dealing with them. With these problems in mind engineers try to borrow ideas from functional programming languages by using the map and reduce primitives as an abstraction that allows to express the simple computations, and hide the complex details of parallelization, fault-tolerance, data distribution and load balancing, hence MapReduce was introduced. The main purposes of this paper are:

- Analyzing MongoDB sharding capabilities
- What is MapReduce and why use it
- Presentation of the results using MapReduce in sharded collections by number of shards used.

In this paper we measure MapReduce time performance through MongoDB, and try to find out how the MapReduce execution time changes with increased number of MongoDB shards. We have described the environment, defined a mini cluster of three virtual machines on which MongoDB is run and we have experimented with a collection of relatively large number of documents. And at the end, the results and conclusions are shown, tried to answer some questions such are the use of MapReduce within MongoDB when is a good option, what needs to be done to speed up the processing and what alternatives to consider.

The rest of this paper is organized as follows: Section 2 presents a summary of some related work in this area. Section 3 contains a short description of the MongoDB where the main point is the shard techniques and possibility of sharding. Section 4 provides the testing results and MapReduce performance evaluation implemented on MongoDB by use different number of shards. Finally, Section 5 provides some conclusions.

II. Related Work

Big companies started facing issues on how to handle the huge amount of data they were receiving and how to process those. Google as the pioneer in search technologies needed computations that process a large amounts of data such as crawled documents, web request logs, graph structure of web documents, etc. According to authors Jeffrey D. and Sanjay G., Google needed a simple solution that was easy to understand, fault tolerant, cheap and reliable. In their paper [1] they analyze MapReduce in large clusters that are highly scalable where hundreds of programs are run and thousands of MapReduce jobs are executed, what is Google on a daily basis.

The authors Smita A. et al., in their paper [2] have introduced an explanation of the MongoDB, its features, advantages and disadvantages. Especially, they address the MongoDB features such as MapReduce, Auto – sharding, MongoDump, etc. They continue with their analyses in case of dealing with small and large amount of unstructured/semi structured data and at the end the conclude that if the amount of the data is big and permanently increases, and high performance and availability are required then MongoDB should be considered as options to use as database.

The authors Zeba Khanam and Shafali Agarwal, in their paper [3], explore large scale data processing using MapReduce and its various implementations to facilitate the database, researchers and other communities in developing the technical understanding of the MapReduce framework. They continue with exploring different MapReduce implementations; most popular Hadoop implementations and other similar implementations using other platforms and compare those based on different parameters.

The authors A. Elsayed et al., in their paper [4], look back to the MapReduce and try to find out the strengths and

weaknesses, dealing with failures and enhancements that could be made to it. Furthermore, they argue that MapReduce doesn't show the expressiveness of query languages like SQL and it needs improvement of limitations such as collocation of related data, implementing efficient iterative algorithms, and managing skew of data.

Another study shows an attempt to analyze user data with MapReduce in real time [5]. The authors Ian B. and Joe D., in their paper show a system which uses the information state collected during a person-machine conversation and a case-based analysis to derive preferences for the person participating in that conversation. They use MapReduce in their processing model to achieve a near real – time generation of user preferences regardless of total case memory size.

Authors Michael T. G. et al., in their paper [6], study the MapReduce framework from an algorithmic standpoint and demonstrate the usefulness of approach by designing and analyzing efficient MapReduce algorithms for fundamental sorting, searching, and simulation problems.

In time when not only big data but also fast data are exploded in volume and availability, authors Wang L. et. al. in their paper [7], address the key challenges that MapReduce is not well suited for and provide solutions with MapUpdate use which is a framework like MapReduce and specifically developed for fast data.

Into the researches [8]-[10] are analyzed the MongoDB, NoSQL databases and reasoning behind choose of them, query optimizations, and comparisons between NoSQL and SQL databases are shown.

Authors Shuai Z. et al., in their paper [11], analyze the MongoDB clusters and introduce how to partition spatial data to distributed nodes in the parallel environment, using its spatial relationships between features.

Mohan and Govardhan, in the papers [12], [13], have analyzed MapReduce as a paradigm and combine it with online aggregation used in MongoDB. Online aggregation, according to them is useful when the data collected from massive clusters and can be very advantageous when the data are collected and estimated from sensors, various social media or Google search. Combining those two area (MapReduce and Online Aggregation) they introduced a new methodology that uses MapReduce paradigm along with online aggregation.

Dede et al. in their paper [14], have evaluated the combination of the MapReduce capabilities of Hadoop with the schema – less database MongoDB, as implemented by the mongo – Hadoop plugin. This study provides insights into the relative strengths and weaknesses of using the MapReduce paradigm with different storage implementations, under different usage scenarios. They have concluded that, in general, if the workload uses MongoDB as a database that needs to be occasionally used as a source of data for analytics then MongoDB is appropriate solution, however, it is not appropriate when using MongoDB as an analytics platform that sometimes must act like a database. Also, they show that using Hadoop for MapReduce jobs is several times faster than using the built-in MongoDB MapReduce capability and this is due to Hadoop file management system (HDFS).

III. MONGODB

Document oriented databases are designated to work without of use of SQL, and instead of it, they use a different language to communicate. A document can have any number of fields listed in any order, like in a relational database. Unlike to relational databases, a row inside a document oriented database, not need to have the same number and types of fields as any other row inside the database. This is due to the fact that there is no schema that restricts a row to be identical in number and the sequence of fields. While there are many document based databases, MongoDB stands out due to its high performance and ease of setup.

MongoDB as document based database uses BSON to store the data, which is the binary – encoded serialization of JSON format. JSON currently supports the following data types: string, number, Boolean, array and object. BSON supports: string, int, double, Boolean, date, byte array, object, array and others. BSON's only restriction is that data must be serialized in little – endian format. Since BSON is a format that the data are sent/retrieved and stored, there is the need of decode those to text. In an analogy with the relational database, a table into MongoDB is a collection of the documents and a database is a group of collections. A document is the most basic entity where MongoDB stores information, similar to a row inside a table in relational database, except the data structure is schema – less. One of the best features of a document is that it may contain other documents embedded inside.

Indexes in MongoDB work almost the same as in relational databases. MongoDB uses B-Tree to implement the indexes and also allows two – dimensional geospatial indexing which is very useful when dealing with location based services.

A. Sharding

The problem of huge amount of data, MongoDB solves in an effective fashion with use of the horizontal data distribution, known as horizontal scaling. Horizontal scaling is shown as very well solution and means a distributed and balanced work across the machines. This way of work in MongoDB is known with the name sharding. Sharding in MongoDB is designed to partition the database into smaller pieces accommodated to different machines, so that no single machine has to store all the data or handle with all the load. MongoDB handles sharding very easily and transparently which means that the interface for querying a sharded cluster is exactly the same as the interface for a single MongoDB instance.

Usually, there are collection which needs to be together and the others which allow or might be need to be distributed across some machines. So, no all collections need to be sharded, but only some collections that need data to be distributed over some shards to improve read and/or write performance. All un – sharded collections will be held in only one shard that is called primary shard (e.g., Shard A in the Fig. 1). The primary shard can also contain sharded collections.

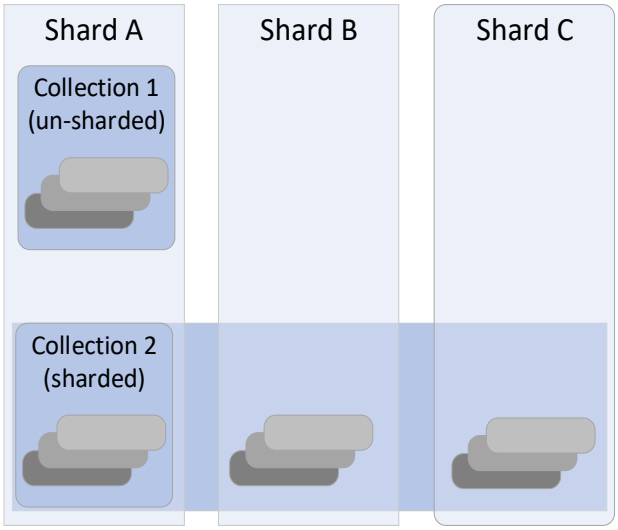

Fig. 1. Example of sharding a collection across multiple shards.

In case of more complex application, we should shard only the collections that would benefit from the added capacity of sharding while leaving the smaller collections unsharded for simplicity. Because sharded and unsharded collections are possible to be accommodated into a same system, all of this will work together, completely transparently to the application. In fact, if later we find that one of the collections that is not sharded, becomes larger and larger, we can shard it, so, it is allowed, at any time, to enable the shard and make a sharding [15].

Manual sharding can be done with almost any database software. Manual sharding is when an application maintains connections to several different database servers, each of which are completely independent. The application manages storing different data on different servers and querying the appropriate servers how to get data back. This approach works well, but there are difficulties when adding or removing nodes to/from the cluster is needed or in face of changing data distributions or load patterns.

MongoDB supports autosharding, and by use of this tries to avoid the abstract architecture from the application and simplify the administration of such a system. MongoDB allows to application to ignore the fact that it isn't talking to a standalone MongoDB server, to some extent. On the operations side, MongoDB automates the data balancing across shards and makes it easier to add and remove capacity.

A sharded cluster consists of shards, mongos routers, and config servers, as shown in Fig. 2.

B. Shared Key

To shard a collection, we have to choose at least one field which will be used to split up the data. This field(s) is called a shard key. In case, when there are a few shards, it's almost impossible to change the shard key, so, it is important to choose a correct one. To choose a good shard key, a good knowledge of the workload and how the shard key is going to distribute the application's requests are needed. And it is often difficult to imagine.

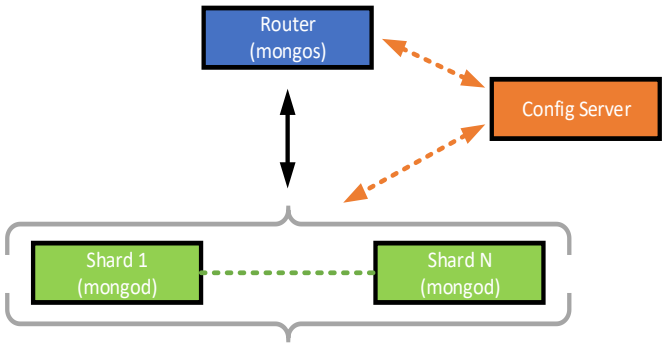

Fig. 2. Components in a Mongodb sharded cluster.

There are three most common distributions ways of splitting the data, which are: ascending key, random, and location – based. Also there are other types (with other key types) but most of those fall into one of the mentioned categories:

Ascending key distribution: The shard key field is usually the data type of Date, Timestamp or ObjectId. With this pattern, all writes are routed to one shard. MongoDB keeps distribution and spends a lots of time migrating data between shards to keep data distribution relatively balanced across the shards. This pattern shows weaknesses in the write scaling.

Random distribution: This pattern is more appropriate in case of when the fields (taken for shard key) do not have an identifiable pattern within dataset. For example, if shard key includes any of the following field username, UUID, email address, or any field which value has a high level of randomness. This is a preferable pattern for write scaling, since it enables balanced distribution of write operations and data across the shards. However, this pattern shows weak performances in case of query isolation, if the critical queries must retrieve large amount of "close" data based on range criteria in which case the query will be spread across the most of the shards of the cluster.

Location – based distribution: The idea around the location-based data distribution pattern is that the documents with some location – related similarity will fall into the same range. The location related field could be postal address, IP, postal code, latitude and longitude, etc.

MongoDB supports three types of sharding strategies:

Range – based sharding: MongoDB divides dataset into ranges determined by the shard key values.

Hash – based sharding: MongoDB creates chunks via hash values it computed from the field's values of the shard key. In general, range – based sharding provides better support for range queries that need query isolation while the hash – based sharding supports more efficiently write operations.

Tag – aware sharding users associate shard key values with specific shards. This type of sharding is usually used to optimize physical locations of documents for location – based applications.

On the below table (Table I) is shown the guidance how to select the shard key.

TABLE I. KEY CONSIDERATIONS FOR A SHARD KEY SELECTION REGARDING THE QUERY ISOLATION AND WRITE SCALING REQUIREMENTS

Query isolation importance	Write scaling importance	Shard Key Selection
High	Low	• Range shard key. • If the selected key does not provide relatively uniformly distribution of data, we can either use a compound shard key or add a special purpose field to our data model that will be used as a shard key. Or for location – based applications we can manually associate specific ranges of a shard key with a specific shard or subset of shards.
Low	High	• Hashed shard key with high cardinality. • If a selected key does not provide relatively uniformly distribution of data, we can add a special purpose field to our data model that will be used as a shard key.
High	High	• A shard key enabling mid – high randomness and relatively uniformly distribution of data. • Determine which shard key has the less performance effect on the most critical use cases. • Special purpose field to our data model that will be used as a shard key.

C. MapReduce

MapReduce is a programming model which is capable to process a huge amount of data with a parallel and distributed algorithm on a cluster. It is a programming paradigm that allow for massive scalability across hundreds or thousands of servers in a Hadoop cluster. MapReduce also is a powerful and flexible tool for aggregating data, solves some problems that are too complex to express by use the aggregation framework's query language. In our case we use MapReduce with JavaScript as its "query language" to express arbitrarily complex logic.

MapReduce processes different problems across large datasets using a large number of computers (or computing nodes) in parallel. Basically, it takes a set of input key/value pairs and produces a set of output key/value pairs [15] and this operations is executed in three steps: Map is the first step, takes the input pairs and to each node applies the "map" function and finally writes the temporary output. To prevent same data being processed a master node ensures that only one copy of redundant input data is processed. Shuffle is the second step, where the shards redistribute that data based on the output keys and reaches a stage that all data with the same key value belonging to the same shard. And finally, reduce is the final step which takes the shuffled data and processes each group of data per key.

MapReduce uses a finalize function to clean the temporary results and to manipulate with the MapReduce output, which are given from the last reduce phase. The finalize function is called before the MapReduce output is saved to a temporary collection. Returning large result sets is less critical with MapReduce, so the call of the finalize function is a good chance to take averages or remove the temporary or unnecessary information in general [16]. MongoDB allows the user to define which shards will execute the map function, the shuffle and reduce and also we can use the same shards for map function execute and as well as reducer function or define other shards that will do that job.

By default, MongoDB creates a temporary collection while MapReduce processes with the data and the temporary collection name is unlikely chosen from a collection name, but, it is a dot – separated string containing MapReduce, the name of the collection which is in MapReduce process, a timestamp, and the database job's ID. It looks something like mr.geonames.1525765769.2. MongoDB automatically destroy this temporary collection when the job is finished and /or MapReduce connection is closed. To keep the temporary collection after the job finishing and connection closed we have to set keeptemp in true as an option parameter. In case that the temporary collection is used, MongoDB allows naming the output collection with the out part option, which is a combined name and out string. To address the last issue MongodB contains an optional parameter called as out and which needs to be set to true, if out parameter is set to true, then there is no need to specify keeptemp, since it is implied. Even if a name for the temporary collection is specified, MongoDB again uses the autogenerated collection name for MapReduce further intermediate steps. When the computations have finished, the temporary collection automatically and atomically will be renamed from the autogenerated name to our set or chosen name. This means that if MapReduce is run multiple times with the same target collection, it will never use an incomplete collection in performing operations. The output collection created by MapReduce is a normal collection, which means that there is no problem with doing a MapReduce on it or a MapReduce on the results from that MapReduce.

IV. MAPREDUCE PERFORMANCE ANALYSIS

To analyze the MapReduce performances, used in MangoDB circumstances, we have created a mini cluster of few virtual servers on which is run MongoDB and the geonames database. Geonames database is an open source database and is taken as an example. Geonames database contains detailed information to world countries such are population, size, geolocation, rivers, villages, capital, etc. [17]. It contains around 11 Million records, rendered on tab separated text file. To manipulate on a better way, we converted those data to a csv format, that could easily be exported to mongo. We scaled down the database only to documents with population larger than zero and the number of those documents was 469660. From the csv file we took into consideration and imported only geonameid as id, asciiname, country and population.

Next, we built a sharded cluster to which was run MongoDB 3.2 under Ubuntu 16.04, based on Fig. 2, through three Virtual Machines, named as mongo-c1, mongo-c2, and mogno-c3, one VM for the configuration server and one query server VM. We indexed the id with "hash" that allows us to create a shard key with the hashed id which makes sure the equally distribution of our geoname collection documents. The hostnames and ip addresses of each VM was set as follow:

- mongo-config: 192.168.157.132
- mongo-router: 192.168.157.130
- mongo-c1: 192.168.157.129
- mongo-c2: 192.168.157.128
- mongo-c3: 192.168.157.131

In our tests a simple map function was set, which finds the country code and returns a value of 1, and reducer function which iterate through the values to count the number of documents in the collection which belongs to each country. The number of documents included in our tests was 469660 and the id was used as a shard key to shard the documents to the different shards.

On the above-mentioned architecture, we executed three tests. In our first test we used only one shard (mongo-c1 was used). The number of documents was 469660 and the total import time was 11.28. In the second test, we used the same number of documents but sharded into two shards (in this case was added the second shard mongo-c2). The total import time in this case was 08.25. The collection was sharded successfully and after sharding the achieved distribution was as follow: into first shard (mongo-c1) 234349 documents and into second shard (mongo-c2) 235311 documents. And in our third test we included another shard (mongo-c3), the same number of documents was included but distributed into three shards. For this case the total import time was 8.47 and the collection was successfully sharded as follow 156646 documents into first shard (mongo-c1), 156693 documents into second shard (mongo-c2) and 156321 documents into third shard (mongo-c3). We executed the same MapReduce job (with the same map and reduce functions) three times to each test and the results are shown on Table II.

TABLE II. MAPREDUCE JOB EXECUTION TIME EXECUTED ON ONE, TWO AND THREE SHARDS USED. NUMBER OF DOCUMENTS USED IS 469660

Execution		1	2	3	Average time
Num. Of Shards	Import time				
1	11.280	9.470	9.238	9.878	9.529
2	8.250	5.773	5.771	5.800	5.781
3	8.470	4.848	4.874	4.922	4.881

To better express the dependence between MapReduce job execution time and the number of nodes used, so the dependence on the number of shards to which the documents are distributed, on Fig. 3, the curve which clearly expresses the decrease of the time with increasing the number of shards is shown.

Next, we again performed the last test, but this time with a little complex shard key. We chose the pair (id, population) as a shard key. Total import time was 8:08. Since the shard key cannot be changed afterwards, so, we drop the before collection shards and recreate a new shard by use of the new shard key. By use of the new shard key the sharding was 349699 documents to the first shard, 119947 documents to the second shard and only 14 documents to the third shard. So, it is produced ununiform distribution. We executed the same MapReduce job as in previous tests and the results are shown on the following table (Table III).

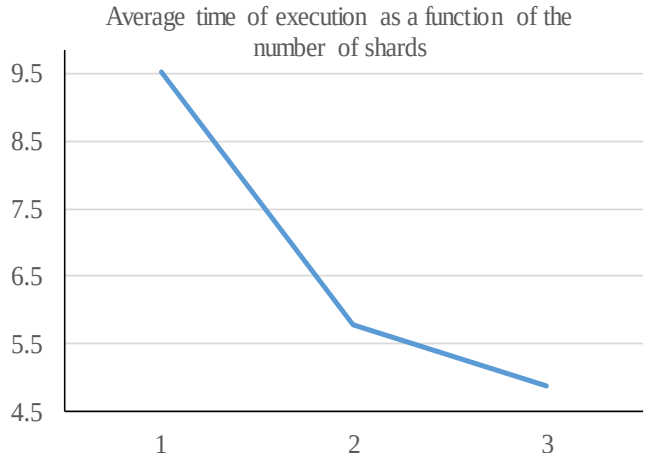

Fig. 3. Average time of mapreduce job execution per number of shards. executed on 469660 documents..

TABLE III. MAPREDUCE JOB EXECUTION TIME EXECUTED ON THREE SHARDS USED. NUMBER OF DOCUMENTS USED IS 469660, SHRD KEY (ID, POPULATION)

Execution		1	2	3	Average time
Num. Of Shards	Import time				
3	8.470	6.685	6.841	6.638	6.721

Regarding to the above tests, clearly we can conclude that when the number of shards increases MongoDB MapReduce performs better and faster. The only trouble as shown in last test is that we should be very precautious in chose of the shard key, so, we need to choose an appropriate (a good shard key which provides as far as possible uniform documents distribution) that will not slow down MapReduce.

V. CONCLUSIONS

Big data has indeed reshaped the way we deal with data. The problems that arise when trying to manipulate huge amounts of data are growing every day and solutions are found from both scientists and companies alike. MongoDB and other NoSQL databases alike has seen growth by providing an alternative to SQL databases. Their design, high availability and fault tolerance have attracted usage in projects where SQL databases cannot be used such as handling unstructured raw huge amount of data.

MapReduce as a framework is designed to solve many problems with huge amount of data, so, MapReduce has a little significance when dealing with small data, but it has an impact when the collections grow. Also it is clear and our tests show that as the number of shards scales up, MapReduce jobs are executed faster especially if we take precautions and use a good shard key. However, the Mongo 3.2 documentation [18] recommends the avoid of the MapReduce use and instead of MapReduce the Aggregation Pipelining is preferred for better performance.

REFERENCES

[1] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), pp. 107-113

[2] Smita Agrawal, Jai Prakash Verma, Brijesh Mahidhariya, Nimesh Patel and Atul Patel. Survey on Mongodb: An Open-Source Document Database. International Journal of Advanced Research in Engineering and Technology, 6(12), 2015, pp. 01-11.

[3] Zeba Khanam and Shafali Agarwal, MapReduce Implementations: Survey and Performance Comparison, International Journal of Computer Science & Information Technology (IJCSIT) Vol 7, No 4, August 2015, pp. 119 - 126

[4] A. Elsayed, O. Ismail, and M. E. El-Sharkawi, MapReduce: State-of-the-Art and Research Directions, International Journal of Computer and Electrical Engineering, Vol. 6, No. 1, February 2014.

[5] Beaver, I. and Dumoulin, J., Applying mapreduce to learning user preferences in near realtime. In: Case-Based Reasoning Research and Development, ICCBR 2014, Berlin, Springer (2014) pp. 15–28.

[6] M. T. Goodrich, N. Sitchinava, and Q. Zhang. Sorting, searching, and simulation in the MapReduce framework. Proceedings of the 22nd international conference on Algorithms and Computation (ISAAC'11), 2011, Takao Asano, Shin-ichi Nakano, Yoshio Okamoto, and Osamu Watanabe (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 374-383.

[7] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri and A. Doan, Muppet: MapReduce-style processing of fast data, Proceedings of the VLDB Endowment, Volume 5 Issue 12, August 2012, pp. 1814-1825

[8] Freire, S., Teodoro, D., Wei-Kleiner, F., Sundvall, E., Karlsson, D. and Lambrix, P. (2016). Comparing the Performance of NoSQL Approaches for Managing Archetype-Based Electronic Health Record Data. PLoS ONE 11(3): e0150069. https://doi.org/10.1371/journal.pone.0150069.

[9] Marwa, E. and Jemili, F., Using MongoDB Databases for Training and Combining Intrusion Detection Datasets. International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2017. pp. 17-29, Studies in Computational Intelligence, vol 721. Springer, Cham.

[10] Parker, Z., Poe, S. and Vrbsky, S., Comparing NoSQL MongoDB to an SQL DB. In Proceedings of the 51st ACM Southeast Conference (ACMSE '13). ACM, New York, NY, USA, Article 5, 6 pages. 2013, DOI: https://doi.org/10.1145/2498328.2500047.

[11] Shuai Z., Bolei Z., Zhenjie C., Sanglu L, "Point collection partitioning in MongoDB Cluster", Research Foundation of Graduate School of Nanjing University (2013CL09), http://www.geog.leeds.ac.uk/groups/geocomp/2013/papers/97.pdf

[12] B. Rama Mohan, A Govardhan, Online Aggregation Using MapReduce in MongoDB, International Journal of Advanced Research in Computer Science and Software Engineering, Volume 3, Issue 9, September 2013, pp. 1157-1165

[13] B. Rama Mohan, A Govardhan, Sharded Parallel Mapreduce in Mongodb for Online Aggregation, International Journal of Engineering and Innovative Technology (IJEIT), Volume 3, Issue 4, October 2013, pp. 119-127

[14] Elif Dede, Madhusudhan Govindaraju, Daniel Gunter, Richard Shane Canon, and Lavanya Ramakrishnan. 2013. Performance evaluation of a MongoDB and hadoop platform for scientific data analysis. In Proceedings of the 4th ACM workshop on Scientific cloud computing (Science Cloud '13). ACM, New York, NY, USA, pp. 13-20.

[15] Li, Feng & Chin Ooi, Beng & Özsu, M. Tamer & Wu, Sai. (2013). Distributed Data Management Using MapReduce. Journal of ACM Computing Surveys (CSUR) Volume 46 Issue 3, January 2014, Article No. 31.

[16] Kyle Banker, MongoDB in Action, 2nd ed., Manning Publications Co. 2016, pp. 333-375.

[17] http://www.geonames.org/, Retrieved 2018-01-31

[18] MongoDB. https://docs.mongodb.com/manual/ , Retrieved 2017-12-16